NAIC-ID(RS)T-0025-95

NATIONAL AIR INTELLIGENCE CENTER

DECOMPILE PROGRAM GRAPH DESIGN AND CONTROLLING FLOW ANALYSIS

by

Lu Jiquan, Hou Wenyong

HUMAN TRANSLATION

NAIC-ID(RS)T-0025-95 1 May 1995

MICROFICHE NR: 95C000276

DECOMPILE PROGRAM GRAPH DESIGN AND CONTROLLING FLOW ANALYSIS

By: Lu Jiquan, Hou Wenyong

English pages: 14

Source: Jisuanji Gongcheng, Vol. 18, Nr. 6, 1992; pp. 33-37

Country of origin: China
Translated by: SCITRAN
F33657-84-D-0165
Requester: NAIC/TATA/Keith D. Anthony

GRAPHICS DISCLAIMER

All figures, graphics, tables, equations, etc. merged into this translation were extracted from the best quality copy available.

DECOMPILE PROGRAM GRAPH DESIGN AND CONTROLLING FLOW ANALYSIS

/38*

Lu Jiquan Hou Wenyong

ABSTRACT

Decompiling is a type of tool capable of taking low level languages and translating them into high level languages. This article stresses the introduction of program graph methods associated with decompile code storage as well as--on the basis of programming graphs--the carrying out of controlling flow analysis and controlling structure restoration with regard to code programs. It gives algorithms for restoring structures and translation of program graphs during controlling flow analysis processes.
KEY WORDS Decompile Controlling Flow Program graph

1 PROGRAM GRAPH DESIGN

Among high level languages, there are several types of control statements, for example, if, while, dowhile, and switch statements, etc. They are capable of insertion into sets with each other, thus making high level languages possess multitudinous variations. Through post-decompilation target codes, these control statements are then implicitly contained in compilation codes or target operating codes, forming complicated, mutually inserted sets of control structures. Among these, there will appear a number of conditional direction change commands. As far as control structure restoration is concerned, it is nothing else than taking these implicitly contained basic control structures and restoring them to shed light on higher level language control statements. This only requires carrying out controlling flow analysis on program code.

---

* Numbers in margins indicate foreign pagination. Commas in numbers indicate decimals.

During controlling flow analysis processes, a good number of intermediate results will be produced. This is due to controlling flow analysis of compilation programs or target operating programs. It is necessary to carry out repeated scanning of program codes and analyze them step by step. Of necessity, intermediate results associated with numerous levels will be produced. The normal display is a type of program form which is higher than compilation language and lower than high level language, that is, a type of transitional result associated with compilation to a high level language. Sometimes, it is possible to call it intermediate language. In order to show this type of multilevel transitional program code, we designed a type of program interior storage form. It satisfies the need to show intermediate products associated with different levels of decompilation, thus flexibly actualizing translations associated with intermediate results between various types of levels.

The storage form associated with this type of program code intermediate product, in actuality, is a type of program flow graph. Using graphic forms to store decompilation results is unusually appropriate to the internal storage of machines. At the same time, during decompilation operating processes, it is necessary to carry out depth prioritized searches. Graphic storage forms are unusually advantageous to this type of search. Fig.1 shows the basic junction point structure associated with program graphs.

What follows introduces several key terms associated with the analysis of flow control:

(1) char tine[40]: storage level no. character string

(2) char opera[10]: operation character

(3) int var[4]: operation no.

(4) int true, false: export outlet

(5) int *ent-1st: previous item interface link

(6) int order: interface order

(7) short ent: previous item interface no.

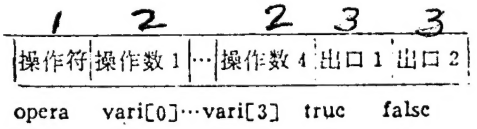
(8) unsigned Nod-flag: information flags associated with relevant interfaces

/34

```
typedef struct
  {
  char line [40];
  char opera [10];
  unsigned char op_type[4];
  un signed short stat[4];
  int vari[4]
  unsigned short used_by, used_of [15];
  int true, false;
  inf *ent_lst;
  int order;
  short ent;
  short Le_c;
  unsigned nod-flag;
  }TREE_NND;
```

Fig.1

Emphasize a look at the functions of operation characters, operation numbers, and export outlets.



Key: (1) Operation Character (2) Operation No. (3) Export Outlet

The meaning is as follows:

(1) vari[3]:=vari[O][opera]vari[1]

This refers mostly to forms of expression.

When opera is not a conditional direction change operation number, false refers to a unique follow-on interface associated with the interfaces in question. When opera is a conditional direction change, ture refers to follow on interfaces associated with conditions that are true. false refers to follow on interfaces that are associated with conditions that are false.

(2) Statements Primarily Aimed at High Level Languages

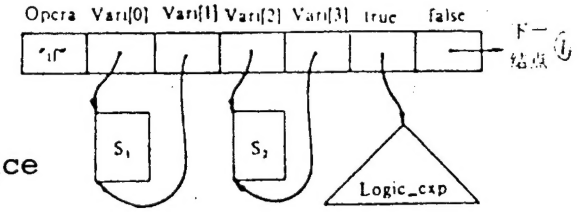
opera expresses a certain statement pattern in high level language--if statements, cyclical statements, switch statements, and so on. Here, as far as cyclical statements are concerned, there are while and dowhile. However, for statements are not included because for statements in this article are taken and turned into the two previous types of cyclical statements.

At this time, this basic interface expresses one statement in high level language. The reason for this is that, in high level language, statements are capable of insertion into sets. Therefore, what is reflected here is a tree of which this basic interface is the root. With regard to language inserted in sets, it is nothing else than sub-trees included in the tree.

This type of basic junction point is appropriate for use in the display of various kinds of intermediate results. During control flow analysis processes, the various control structures, in all cases, use this interface for display.

(1) if-else statement

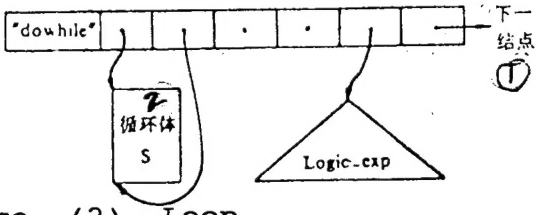
if (logic-exp) then s_1 else s_2



Key: (1) Next Interface

(2) do-while statement

do S while (logic-exp)

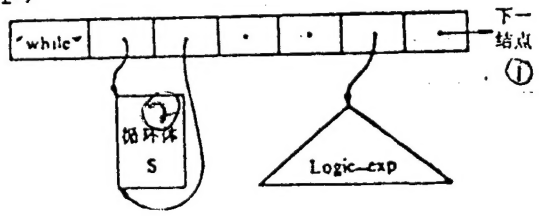


Key: (1) Next Interface (2) Loop

(3) while statement

while (logic-exp)

S



Key: (1) Next Interface (2) Loop

(4) switch statement

```
switch (exp)
{case C0: S0
case C1: S1
   .
   .
   .
case Cn: Sn
```

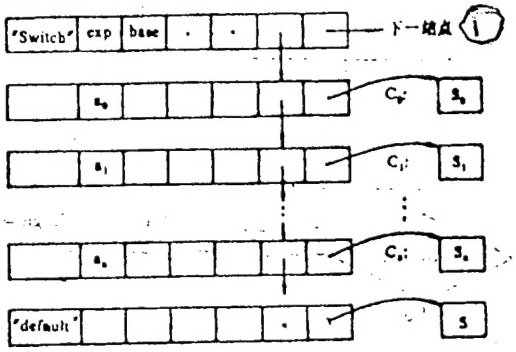
```
        default: S
        }
```

In the Fig. below, Ci=base+ai(i:0~n)

C_i is a constant

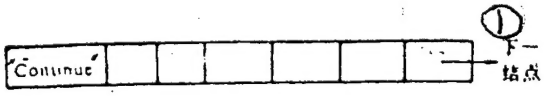
base is a base constant

a_i is an integer serial no.



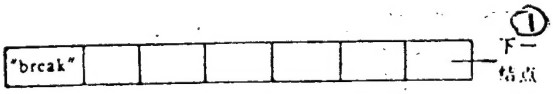
Key: (1) Next Interface

(5) continue statement



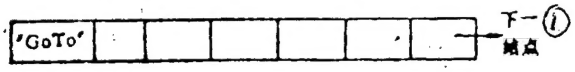
Key: (1) Next Interface

(6) break statement



Key: (1) Next Interface

(7) GoTo statement



Key: (1) Next Interface

(8) Formulaic Statements

Formulaic statements are generally used to express tree forms (illustrate with examples).

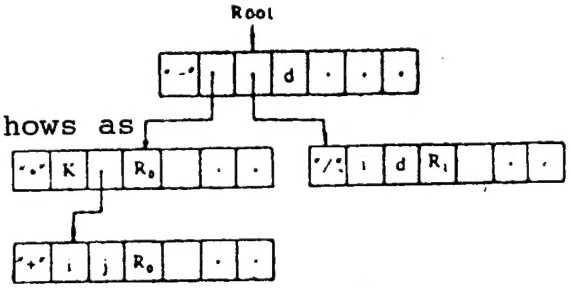
This is due to formulaic expressions being capable, during multiple operations--more than 3 operation numbers--of exceeding basic interface expression capabilities.

For example, the expression d=(i+j)*k-i/d

The corresponding compilation language program section is:

```
ADDL, i, j, R.
MuLL k, R.
DIVL i, d, Ri
SUBL R0, Ri d
```

Using the expression form tree, it shows as



On the basis of middle level universal mechanisms, this tree then arrives at this form of expression.

In the expression forms above, S_0, S_1, S_2,...S_n, S, are all program sub-graphics formed using basic interfaces. In reality, various types of statements are formed into a tree using basic junction points. When decompilation operations reach a certain stage, controlling flow analysis is complete. The entire program graph is then one regressive tree, appearing as a single linked form. The first junction point interface is none other than the first statement in the program. Each interface expresses one statement in high level language. Moreover, each junction point is a tree using interfaces as roots in order to express a

statement. Each interface in trees expresses one statement in the language. Moreover, the mutual relationships between junction points embody the relationships associated with mutual set insertion between various statements in high level languages. The entire program graph is nothing else than a graphical expression of high level programs, capable of being turned directly into high level target programs.

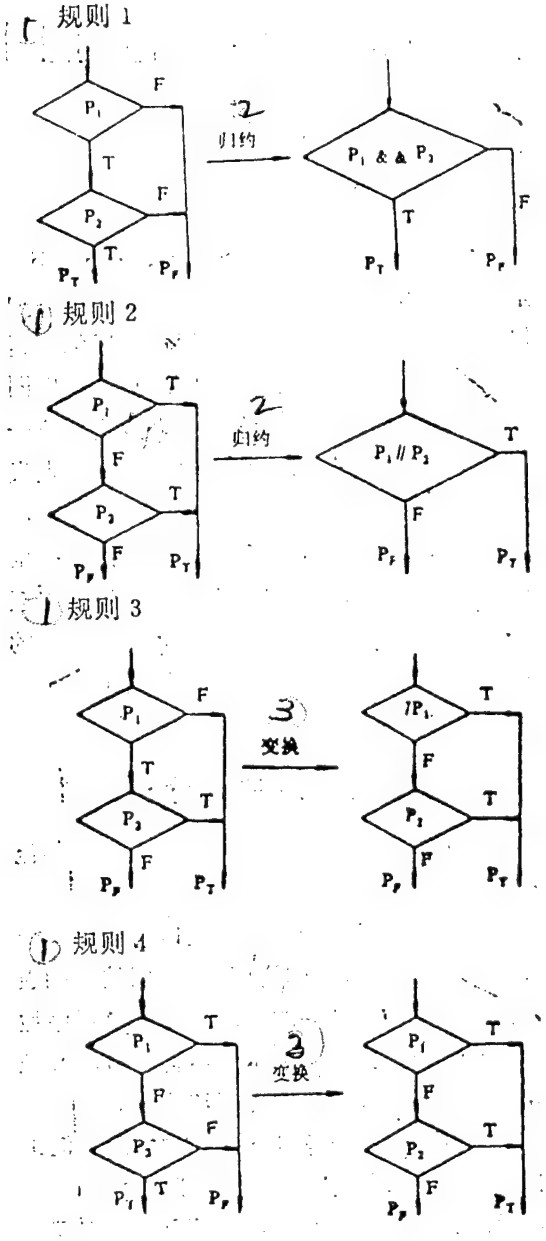
2 CONTROLLING FLOW ANALYSIS BASED ON PROGRAM GRAPHS

Below, control flow analysis is carried out on code programs based on program graphs.

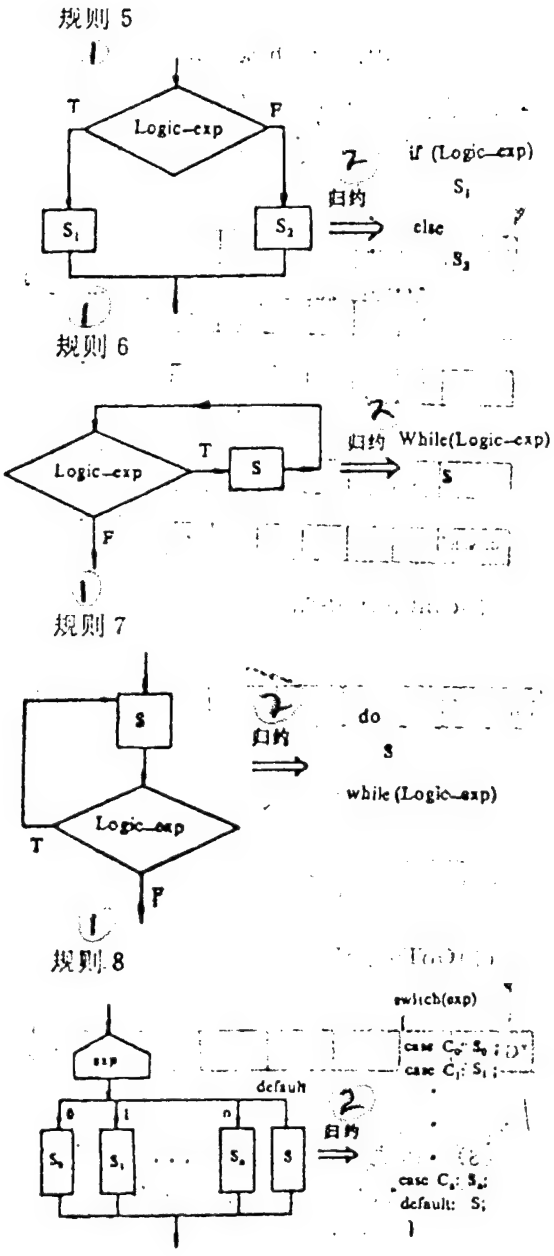
Various types of control structures associated with compilation languages are expressed by the use of conditional or nonconditional translations. This is one form of expression for weak linear structures. Program control structure recovery is nothing else than the need to carry out analysis of program controlling flow, seeking out the basic structural component--in conjunction with this, taking it and shedding light on equivalent high level language control structures.

What must be carried out first is the recovery of logic expressions. Compilation language--when carrying out logic decisions--operates relying on machine status registers. Due to the fact that status registers are only capable of recording one iteration of logic comparison results--when determining whether or not there are multiple conditions--compilation language, therefore, is only capable of adding processing to these conditions one at a time. In conjunction with this, lateral logic direction change connections are used in order to express the logical relationships between various decisions. The summation of these logical decisions is, at the same time, the first step associated with the recovery of program control /36 structures. Logic expressions obtained from merging are nothing else than logical expressions in high level languages--appearing in else statements, switch statements, and cyclical statements.

The production of logical expressions is based on using the four rules below. Among these, two are translation rules. Two are induction rules.



Key: (1) Rule (2) Induction (3) Translation



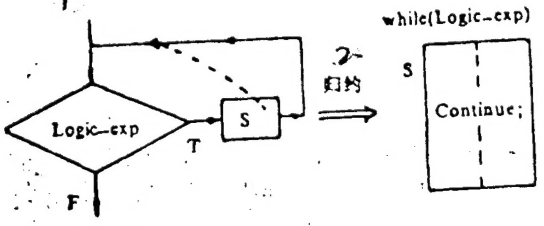
Key: (1) Rule (2) Induction

With regard to program graphs repeatedly carrying out merging--that is, actions on program graphs in accordance with the translation rules and induction rules above--recursive operations are carried out until it is not possible to translate or induce anymore, and they stop. In this way, logic expressions are then restored. As far as carrying out analysis on these logic expressions and program graphs is concerned, it is then possible to restore control structures associated with high level language. Restoration of control structures has the following four rules.

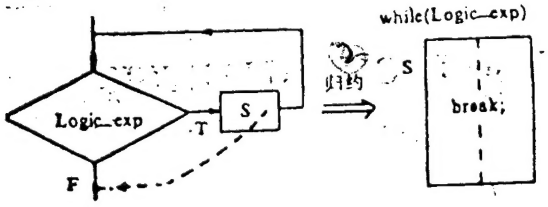
Continue and break statements in C language--speaking in fundamental terms--are GoTo statements. The two statements make C language more flexible to control cycles while making C programs more succinct and easy to understand--embodying C language characteristics. However, as far as these two sentences are concerned--while making C language program flow graphs give rise to the appearance of unstructured factors--break statements will cause cyclic abnormality exports. Continue sentences create coordinate cycles. If GoTo sentences are used for the realization of the unstructured portion of program flow graphs given rise to by continue and break, then, with regard to /37 original programs, very large changes are produced. Readings can vary, bringing difficulties for understanding programs. What is most important is that there is no embodiment of the characteristics of C language itself. Because of this, in C decompilation, recovery of Continue and break is extremely important.

Restoring--in C language--break statements and Continue statements is based on the rules below:

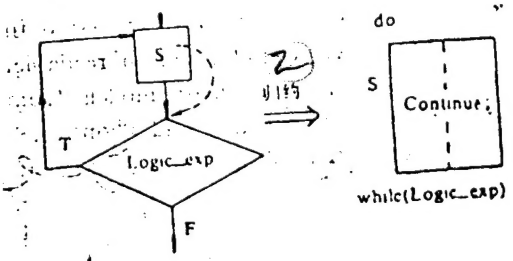
规则 9



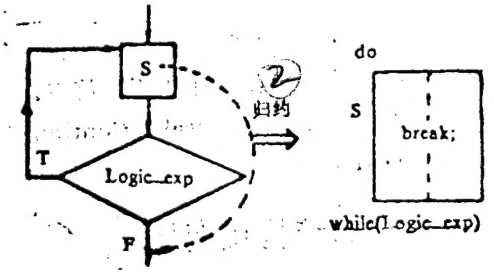
规则 10



规则 11



规则 12



Key: (1) Rule (2) Induce

As far as the restoration of program flow graph control structures are concerned, it is nothing else than the carrying out of repeated scanning of program graphs and step by step analysis. Periods produce a good many intermediate results. We take these intermediate program codes and store them in program graphs, that is, transformations associated with each iteration of intermediate results, reflecting--in program graphs--alterations associated with interface junction point information. Through repeated scanning, step by step analysis, and restoration, program graphs also follow along with translation and gradually approach the control structures of high level language. When control flow analysis reaches a certain stage, that is, program graphs no longer translating, then, the whole analysis process is finished. Program graphs at this time come close to expressing high level programs which are translated out.

Below, basic algorithms for control structure restoration are introduced:

(1) With regard to logic decisions, carry out merging, producing logic expressions.

(2) Identify and record cyclical structures.

(3) Identify and restore break statements and continue statements in cyclical structures as well as GoTo statements which jump out from within cycles.

(4) Identify if-else structures and Switch structures.

(5) Carry out translations on identified if-else structures and switch structures, making them able by induction to become if-else structures and Switch structures. At the same time, identify unstructured components.

(6) Carry out processing of unstructured components.

(7) Restore various types of control structures, taking results and storing them in program graphs.

Due to mutual insertion into sets associated with various types of control structures, each step in algorithms, therefore, is capable in all cases of needing recursive transfers. What corresponds to this is nothing else than the carrying out of

repeated scanning of program graphs. Again, taking each scanning iteration result to store in program graphs, it causes program graphs to translate step by step into high level programs.

Our decompilation system is already realized on Micor VAX II, under VMS operating systems, using C language. At the present time, this system's operating results are very good.

REFERENCES

1 唐稚松、郝克刚. 计算机软件开发方法、工具和环境. 西安: 西北大学出版社

2 Hopwood, gregory Littell 1978 Decompilation, PH. D dissertion

3 Mosgol B., Tool Ada and the "Middle End" of the PQCC Ada Compiler 1980

(Editor Zhang Tingjun)

DISTRIBUTION LIST

DISTRIBUTION DIRECT TO RECIPIENT

ORGANIZATION	MICROFICHE
B085 DIA/RTS-2FI	1
C509 BALLOC509 BALLISTIC RES LAB	1
C510 R&T LABS/AVEADCOM	1
C513 ARRADCOM	1
C535 AVRADCOM/TSARCOM	1
C539 TRASANA	1
Q592 FSTC	4
Q619 MSIC REDSTONE	1
Q008 NTIC	1
Q043 AFMIC-IS	1
E051 HQ USAF/INET	1
E404 AEDC/DOF	1
E408 AFWL	1
E410 AFDTC/IN	1
E429 SD/IND	1
P005 DOE/ISA/DDI	1
P050 CIA/OCR/ADD/SD	2
1051 AFIT/LDE	1
P090 NSA/CDB	1
2206 FSL	1

Microfiche Nbr: FTD95C000276
NAIC-ID(RS)T-0025-95